Source: cppcc docs/MeTTa_Bulk_Load_Benchmark_Draft_0.1.md @ 934786a — curated 2026-08-08; edits land in the source first.

MeTTa bulk load — a benchmark

What it costs to produce a CCS binary once, and to read it many times

Draft 0.1 — 2026-08-08

Measurements of loading MeTTa sources through the Hyperon reference implementation and through a CCS-generated frontend. The headline is an asymmetry rather than a speedup: producing a compiled binary is expensive and happens once; reading it is cheap and happens as often as you like.

Every figure here comes from a single measurement run and is reproducible from the published code.

---

1. What was measured

Three MeTTa corpora, seven timed regimes, 100 iterations each, median microseconds on one machine.

corpus	bytes	atoms
small.metta	187	13
medium.metta	24 789	146
large.metta	140 931	3 216

The regimes, stated precisely — the comparison is only as honest as these definitions:

regime	what is timed
PY	reading the file into memory — I/O baseline only

```
| MD        | hyperon MeTTa.run(text): parse + EVAL + insert        |
| MDparse   | hyperon MeTTa.parse_all(text): parse only             |
| MAI       | atom insertion alone, atoms pre-built outside the     |
|           |   timed region                                        |
| WK        | walking a compiled .bfgr (output discarded)           |
| WK+MAI    | THE CONSUMER COST: walk the binary, insert the atoms  |
| MP        | THE PRODUCER COST: emit the .bfgr from source         |
| WK+MP     | produce and then consume, in one figure               |
```

`WK+MAI` is reported for two consumer implementations — a Python one and a Rust one — because the consumer language dominates this cost.

---

2. The measurements

corpus	PY	MD	MDparse	MAI	WK	WK+MAI	MP
small.metta	6	74	78	8	62	5.38	1 985
medium.metta	8	452	2 019	505	1 210	259	6 468
large.metta	65	5 917	25 048	3 557	11 391	1 584	29 498

median µs, 100 iterations; WK+MAI is the Rust consumer (Python figures in §5)

---

3. The headline: an asymmetry, not a speedup

The number worth understanding is not "CCS is N× faster". It is that the work divides into a cost paid **once** and a cost paid **per read**, and those costs differ by more than an order of magnitude.

Take `large.metta`. Producing the binary costs **29 498 µs**, once. Reading it afterwards costs **1 584 µs**, every time. Running the reference implementation costs **5 917 µs**, every time, because it re-does the work on each load.

So for *N* loads:

```
reference implementation :  N × 5 917
CCS (produce, then read) :  29 498 + N × 1 584
```

Those cross at **N ≈ 7**. Below seven loads, compiling first is not worth it. Above seven, every additional load is 3.7× cheaper, and the one-time cost recedes.

Against the parse-only regime the crossover is **N ≈ 2**.

This is the shape that matters for a knowledge store, where sources are loaded far more often than they change. It is also why a single speedup figure would misrepresent the result: at N = 1 the reference implementation wins outright.

---

4. Per-load ratios, both ways

Two ratios can be formed against the reference implementation, and they differ substantially. Both are given, because presenting only one would be choosing the flattering comparison.

corpus	MD / (WK+MAI_rust) vs parse+eval+insert	MDparse / (WK+MAI_rust) vs parse only
small.metta	13.8×	14.5×
medium.metta	1.75×	7.8×
large.metta	3.74×	15.8×

Neither column is the "right" one.

`MD` includes **evaluation**, which the CCS consumer is not doing — so that column charges the reference implementation for work outside the comparison, and understates CCS's disadvantage.

`MDparse` is parse-only and therefore closer to like-for-like — but it is measured through an interface that materialises every atom, which the `MD` path does not.

The conservative reading is the left column. The like-for-like reading is the right one. A reader who wants a single number should take the left.

⚠ One measured oddity, unexplained

`MDparse` is *slower* than `MD` on medium and large — 2 019 against 452, and 25 048 against 5 917. Parsing alone costs more than parsing, evaluating and inserting.

We do not know why. It is reproducible and it is reported as measured. A plausible guess involves the cost of materialising atoms across the interface boundary, but it is a guess and is not offered as a finding. Anyone drawing conclusions from the `MDparse` column should know this.

---

5. Per-atom rates

Normalising by atom count on `large.metta`, comparing the two consumer implementations:

```
┌──────────┬────────────────┬──────────────┬──────────┐
│ regime   │ Python         │ Rust         │ ratio    │
├──────────┼────────────────┼──────────────┼──────────┤
│ walk     │ 3 543 ns/atom  │  301 ns/atom │   11.8×  │
│ insert   │ 1 106 ns/atom  │  269 ns/atom │    4.1×  │
│ combined │ 4 647 ns/atom  │  493 ns/atom │    9.4×  │
└──────────┴────────────────┴──────────────┴──────────┘
```

The consumer language is the dominant variable in these numbers. A Python consumer erases most of the advantage of §3; a Rust one keeps it. That is a fact about consumers, not about CCS, but it decides whether the approach pays.

---

6. Why these numbers replaced an earlier set

An earlier run reported a considerably better result — roughly 10× where §4 now reports 3.74×.

That run was wrong, and it was wrong in a way worth describing. The MeTTa tokenizer had a defect that silently dropped content containing apostrophes. Fewer atoms were produced, so less work was measured, so CCS looked faster. The atom counts did not match the reference implementation's, which is how it was caught:

```
                   reference    before fix    after fix
small.metta            13           13           13   ✓
medium.metta          146          109          146   ✓
```

The defect was fixed and every figure re-measured. The numbers in this document are the post-fix ones, and they are less impressive than the numbers they replace.

Two reasons to publish this rather than quietly ship the better figure. It explains why an atom-count check belongs in any benchmark that compares parsers — a silently truncated parse looks exactly like a fast one. And a benchmark whose authors caught themselves is worth more than one whose authors did not look.

---

7. What this does not establish

- **The corpus is small.** The largest input is 141 KB and 3 216 atoms. These are not large-knowledge-base numbers, and nothing here should be extrapolated to one.
- **One machine, one run.** Median of 100 iterations, no cross-machine variance, no confidence intervals.
- **The walk regime uses a proxy.** `WK` is measured through an existing structure-walking path rather than a purpose-built consumer; a real consumer doing real work would cost more.
- **The Python figures are older than the Rust figures.** Python regimes were carried forward from an earlier run; Rust regimes were re-measured after the fix of §6. They are compared here as though contemporaneous.
- **Correctness is a separate question.** Nothing in this document says the generated frontend agrees with the reference parser on the corpus. The conformance position is set out in *MeTTa on CCS* §3, and it is a harness with an enumerated divergence list, not proven equivalence.

References

- *MeTTa on CCS* — language support, source positions, conformance.
- *CCS Technical Specification*, Draft 1.0.
- US Patent 8,464,232.
- The `metta_bench` use case, published alongside this document, which carries the benchmark code and corpus.